Real-time Task Scheduling Using Extended Overloading Technique for Multiprocessor Systems*

Wei Sun[1], Chen Yu[1], Yuanyuan Zhang[2], Xavier Defago[1] and Yasushi Inoguchi[1, 3]
[1]Graduate School of Information Science,
[3]Center for Information Science,
Japan Advanced Institute of Science and Technology,
1-1, Asahidai, Nomi, Ishikawa, 923-1292 Japan
{sun-wei, yuchen, defago, inoguchi}@jaist.ac.jp

[2] Fujitsu Laboratories Ltd.
Kawasaki, Kanagawa, 211-8588, Japan
zhang.yuanyuan@jp.fujitsu.com

Abstract

The scheduling of real-time tasks with fault-tolerant requirements has been an important problem in multiprocessor systems. Primary-backup (PB) approach is often used as a fault-tolerant technique to guarantee the deadlines of tasks despite the presence of faults. In this paper we propose a PB-based task scheduling approach, wherein an allocation parameter is used to search the available time slots for a newly arriving task, and the previously scheduled tasks can be rescheduled when there is no available time slot for the newly arriving task. In order to improve the schedulability we extend the existing PB-overloading and the Backup-backup (BB) overloading. Our proposed task scheduling algorithm is compared with some existing scheduling algorithms in the literature through simulation studies. The results have shown that the task rejection ratio of our real-time task scheduling algorithm is lower than the compared algorithms.

1. Introduction

In a real-time multiprocessor system, fault-tolerance can be provided by scheduling multiple copies of tasks on different processors [1-8]. Primary-backup based scheduling is one of fault tolerant scheduling techniques. In the PB-based task scheduling two versions of a task, primary version and backup version, are scheduled on two different processors and the acceptance test is used to check the correctness of the execution result [4-8].

In order to improve the schedulability, overloading techniques are often used. PB-overloading is defined to schedule the primary of a task onto the same or overlapping time slot with the backup of another task on a processor [8]. BB-overloading is defined to schedule the backups of multiple tasks onto the same or overlapping time slot on a processor [4, 7, 8]. In [8], R. Al-Omari et al. drew a conclusion that the PB-overloading is able to achieve better performance than BB-overloading, and BB-overloading algorithm is better than no-overloading algorithm.

In this paper, we address a PB-based scheduling of non-preemptive aperiodic real-time tasks with fault-tolerant requirements. In this PB-based scheduling, both PB-overloading and BB-overloading exist, and an extended overloading strategy is used to make the overloading more flexible and efficient. Our scheduling algorithm can reschedule the previously scheduled tasks on one processor. For simplicity, we assume that, at any time, at most one single processor can be crashed. In other words, we consider 1-timely-fault tolerant schedules, where a *k*-timely-fault-tolerant (*k-TFT*) schedule is defined as the schedule in which no task deadlines are missed, despite *k* arbitrary processor failures [10]. The objective of the paper is to decrease task rejection ratio.

2. Related work

In PB-based task scheduling a backup is deallocated when its primary is finished successfully [4, 6, 8]. In [7], resource reclaiming, which refers to the problem of utilizing resources left unused by a task version [11], is used to improve the processor utilization. Thus there might be some empty time slots in history schedules due to the resource reclaiming. The empty time slots should be reused by new tasks.

Backups are scheduled as late as possible or overloaded on other backups as much as possible, and a function is used to control the overlapping length between overloaded backups in [4]. When scheduler can not find a proper time slot for a new task, a primary will be rescheduled by moving it forward while any backups can not be rescheduled. However, sometimes it is necessary to move tasks backward.

In [6, 8] the scheduling algorithms are based on the Spring scheduling approach [9], which is a heuristic algorithm and dynamically schedules tasks with resource requirements. The algorithms in [2, 3, 6-8] can not reschedule tasks.

In [8], PB-overloading chain will not contain more than two tasks at the same time, for example the chain *A* in Figure 1. But, in theory, as long as the time between the first task and the last task in the PB-overloading chain is

*This research is conducted as a program for the "21st Century COE Program" by Ministry of Education, Culture, Sports, Science and Technology, Japan

1550-6525/07 $25.00 © 2007 IEEE
DOI 10.1109/DS-RT.2007.12

less than the minimum time interval of faults, a PB overloading chain can contain more than two tasks, for example the chain *B* in Figure 1, for a PB-overloading chain can only tolerate one failure [8]. Moreover, PB-overloading chain should be opened but looped, and the looped PB-chain will fail eventually, for example the looped chain *C*, will fail, if Processor 4 fails.

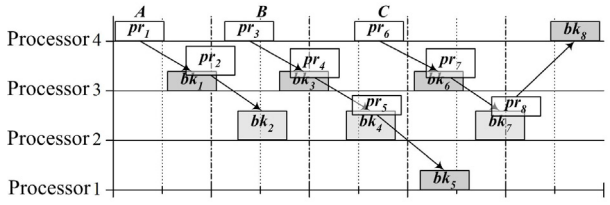


Figure 1: PB-overloading chains.

Considering these existing problems, our task scheduling algorithm can reschedule primary and backup tasks by moving them forward or backward within the reasonable scope. Because of the large time cost to reschedule tasks on all processors, the rescheduling only takes place on one processor and the relationship of overloaded tasks can not be changed. The two overloading techniques are extended to contain more tasks and can co-exist in our algorithm.

3. Models

3.1 Scheduler model

The scheduler model used in this paper is similar with those in [2, 6-8]. All processors have identical computing capability and are connected through a shared medium. The scheduler is running in parallel with the processors. Each processor has its own task queue. A tuner is in front of a local processor task queue and in charge of inserting a new task into this task queue or changing the previous schedule. The structure of scheduler is shown in Figure 2. It is assumed that the scheduler has been made fault tolerant by other fault tolerant technique, for example, modular redundancy technique [8].

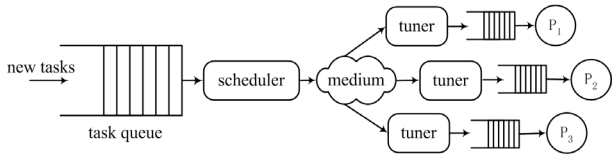


Figure 2: System structure.

3.2 Task model

Tasks have the following attributes:

1. Tasks are aperiodic, i.e., task arrivals are not known in advance. Each task $\boldsymbol{T_i}$ has the numeric characteristics: *arrival time* ($\boldsymbol{a_i}$), *ready time* ($\boldsymbol{r_i}$), *worst case computation time* ($\boldsymbol{c_i}$), *actual computation time* ($\boldsymbol{ac_i}$) and *deadline* ($\boldsymbol{d_i}$). The actual computation time is the true time that a processor takes to finish a task. The worst case computation time is assumed always larger than the actual computation time.
2. Each task has two identical versions. The version to be scheduled earlier in a schedule is marked as primary ($\boldsymbol{pr_i}$) and the other one is marked as backup ($\boldsymbol{bk_i}$). When a primary is finished successfully, its backup will be deallocated at once. The outputs of the primary and its backup are absolutely identical for ever.
3. Tasks are independent and non-preemptable.

3.3 Fault model

Each processor, except the scheduler, may fail due to hardware or software faults which result in task failures. The faults can be transient or permanent. Each fault is independent to the others and exists in one processor.

MTBF is defined to be the expected time between two failures. ***TTSF*** is defined to be the time to the second failure, i.e., the critical time between two failures. The longer *TTSF* means the weaker reliability. The maximum number of processors that are expected to fail at any time point is assumed to be one, because only *1-TFT* is considered in this paper. We also assume that $\forall T_i(d_i - r_i)$ is much less than *MTBF*. If any overloading does not happen, in the worst case, *TTSF* will be equal to $\max(d_i - r_i)$.

A fault-detection is assumed to announce failures in time. The scheduler will not schedule tasks to a known failed processor.

3.4 Definitions

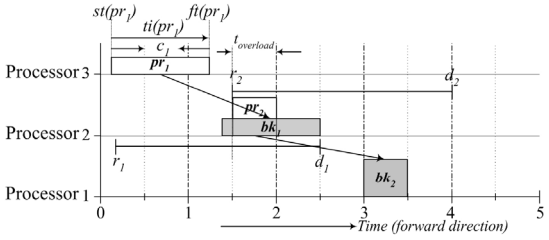
1. $\boldsymbol{st(\cdot)}$ is the start time of pr_i or bk_i. $\boldsymbol{ft(\cdot)}$ is the finish time of pr_i or bk_i.
   Constraint 1: $r_i \le st(pr_i) < ft(pr_i) < st(bk_i) < ft(bk_i) \le d_i$.
2. $\boldsymbol{proc(\cdot)}$ is the processor on which the primary or backup is scheduled.
   Constraint 2: $proc(pr_i) \ne proc(bk_i)$.
3. $\boldsymbol{ti(\cdot)}$ is the time interval from $st(\cdot)$ to $ft(\cdot)$ on which the primary or backup is scheduled.
   Constraint 3: $ti(pr_i) \cap ti(bk_i) = \phi$.
4. $\boldsymbol{n_{cascade}}$ is the cascade number of overloaded tasks within a time slot. $\boldsymbol{m}$ is the number of processors. when $n_{cascade} = 1$, it means the task is scheduled without overloading; when $n_{cascade} = m$, it means no task can be overloaded on this time slot again.

Constraint 4: $1 \le n_{cascade} \le m$.

5. $\boldsymbol{t_{overload}}$ is the part time of a task overloaded on other tasks.
   Constraint 5: $0 \le t_{overload} \le c_i$.
6. A set of tasks which are overloaded with each other within a time slot is named an overloading task set. This task set is denoted as τ. $st(\tau)$ is the start time of the first task to be executed, and $ft(\tau)$ is the finish time of the last task to be finished in τ. A single task is also a task set with only one task.
7. The shift window $\boldsymbol{Win_s(\cdot)}$**< *back, for* >** is the time interval on which a previously scheduled task can move. *for* is the time of a task being moved forward. *back* is the time of a task being moved backward. All tasks in an overloading task set (τ) have the same Win_s, which is the $Win_s(\tau)$.
8. A single PB-chain is defined to be that any primary in this chain exists in only one overloading task set. If a primary in a PB chain and a primary in another PB chain are in the same overloading task set, the two chains are coupled.
9. The maximum space length $\boldsymbol{L_s}$ of a single PB-overloading chain is defined to be the maximum number of primaries. The maximum time length $\boldsymbol{L_t}$ of a single PB-overloading chain is defined to be the time interval between the earliest start time of tasks and the latest finish time of tasks in this chain.

Some definitions are shown in Figure 3. The detailed example of the overloading task set and the shift window will be shown in Section 4.4 with the scheduling algorithm.

In this paper, the overloading techniques are extended. An example of overloading in this paper, in Figure 4, illustrates a single PB-overloading chain and two coupled PB-overloading chains. Chain *A* is a single PB chain. Chain *B* and Chain *C* are coupled on Processor 2. L_t of each chain is shown in the figure. *B* and *C* have the same L_t. When pr_5 is finished, *B* and *C* will be decoupled.



$n_{cascade}$ of pr_1 and bk_2 is 1.
$n_{cascade}$ of bk_1 and pr_2 is 2.
$proc(pr_1)$ is Processor 3.
$Wins(bk2)$=<1,0.5>
In this figure, m is 3.

Figure 3: Illustration of definitions

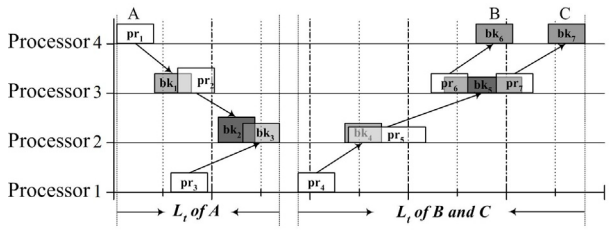


Figure 4: Examples of a single PB overloading chain and two coupled PB overloading chains.

BB-overloading will not increase *TTSF* dramatically. Thus, L_t will decide *TTSF* at most time. Usually *TTSF* is expected to be as small as possible. The smaller the *TTSF*, the better the fault-tolerant technique is. However, as long as the overloading exists, *TTSF* will increase. It is a tradeoff between reliability and schedulability. The scheduler can know the minimum time interval of faults from the history. In this paper, L_t is set simply to be half of the minimum time interval between faults in the system history.

4 Proposed task scheduling algorithm

The proposed task scheduling approach in this paper consists of the validity checking, the chief scheduling algorithm and the rescheduling algorithm. The validity checking module is used to guarantee the allocation and overloading are valid. The chief scheduling algorithm is used when it is easy to find available time slot for new tasks. If the chief scheduling algorithm can not find available time slots for new tasks, then the previously scheduled tasks will be rescheduled.

What time to start the new task and on which processor to allocate the task is a common problem for real-time task scheduling algorithms. In this paper, an allocation parameter (***AP***) is used to evaluate every possible task allocation.

4.1 Allocation parameter (AP)

For a new primary pr_i to be allocated on processor j, $\boldsymbol{AP[pr_i, p_j, ft(pr_i)]}$ of each possible allocation (possible $ft(pr_i)$) is defined as follows:

$$AP\left[pr_i, p_j, ft(pr_i)\right] = \begin{cases} \dfrac{d_i - ft(pr_i)}{d_i - r_i} \cdot \dfrac{1}{m} & \text{for } n_{cascade} = 1, \\ & \text{non-overloading} \\ \dfrac{d_i - ft(pr_i)}{d_i - r_i} \cdot \dfrac{n_{cascade}}{m} \cdot \dfrac{t_{overload}}{c_i} & \text{for } 1 < n_{cascade} \le m, \\ & \text{overloading.} \end{cases} \quad (1)$$

and $\boldsymbol{AP(pr_i, p_j)}$ is defined as:

$$AP\left(pr_i, p_j\right) = max\left\{AP\left[pr_i, p_j, ft(pr_i)\right]\right\}. \quad (2)$$

For a new backup bk_i to be allocated on processor j, $\boldsymbol{AP[bk_i, p_j, st(bk_i)]}$ of each possible allocation (possible $st(bk_i)$) is defined as follows:

$$AP\left[bk_i, p_j, st(bk_i)\right] = \begin{cases} \dfrac{st(bk_i) - r_i}{d_i - r_i} \cdot \dfrac{1}{m} & for\ n_{cascade} = 1, \\ & non-overloading, \\ \dfrac{st(bk_i) - r_i}{d_i - r_i} \cdot \dfrac{n_{cascade}}{m} \cdot \dfrac{t_{overload}}{c_i} & for\ 1 < n_{cascade} \le m, \\ & overloading. \end{cases} \quad (3)$$

and $\boldsymbol{AP(bk_i, p_j)}$ is defined as:

$$AP\left(bk_i, p_j\right) = max\left\{ AP\left[bk_i, p_j, st(bk_i)\right]\right\}. \quad (4)$$

Thus, we define the allocation parameter for a new task:

$$AP = \begin{cases} max\left[AP\left(pr_i, p_j\right)\right] for\ 1 \le j \le m, \\ \qquad when\ a\ primary\ is\ to\ be\ scheduled, \\ max\left[AP\left(bk_i, p_j\right)\right] for\ 1 \le j \le m, \\ \qquad when\ a\ backup\ is\ to\ be\ scheduled. \end{cases} \quad (5)$$

Since we have

$$0 < \frac{st(bk_i) - r_i}{d_i - r_i} < 1, \quad (6)$$

$$0 < \frac{d_i - ft(pr_i)}{d_i - r_i} < 1, \quad (7)$$

$$0 < \frac{n_{cascade}}{m} \le 1, \quad (8)$$

$$\text{and } 0 < \frac{t_{overloading}}{c_i} \le 1, \quad (9)$$

the value of AP is between 0 and 1.

Because of the deallocation of backups, the larger the distance between a primary and its backup, the smaller influence of backups on task rejection is. The primary is scheduled as early as possible that is $st(pr_i)$ tries to be closer to r_i; the backup is scheduled as late as possible that is $ft(bk_i)$ tries to be closer to d_i, i.e., Eq.6 and Eq.7. In order to decrease rejection ratio, an overloading task set is scheduled to contain as many tasks as possible, and the time slot occupied by an overloading task set is scheduled as short as possible. Thus, the larger $n_{cascade}$ and $t_{overload}$, the better schedulability is. Eq.8 and Eq.9 can represent these.

4.2 Validity checking and overloading constraints

The validity checking consists of all constraints mentioned in Section 3.4 and the following overloading constraints.

1. Backups can be overloaded on any task. Primaries can be overloaded only on backups.
2. If a primary pr_i is overloaded on a backup bk_j, $st(pr_i)$ must be later than $ft(pr_j)$.
3. A new task can only be overloaded on one task set.
4. If the number of processors is m, the maximum L_s is m-1.
5. The maximum L_t is equal to or larger than $max(d_i\text{-}r_i)$ and much less than $MTBF$.
6. A single PB-overloading chain should be opened but looped, that is the task sets of a chain can not exist on the same processor (a PB chain can also be looked as a chain of overloading task sets, see Definition 6). A looped chain has been shown in Figure 1.

4.3 Scheduling algorithms

In our scheduling algorithm, a new task T_i will be scheduled on its arrival, i.e., FCFS. The APs of both the primary and the backup of a new task are calculated, and the primary and the backup of this task are scheduled to the corresponding allocations. It is possible that a processor has the same AP value with the other one. If two same AP values exist, the processor on which the task can achieve larger $n_{cascade}$ is selected. If the two $n_{cascade}$ are still identical, the processor on which the task can achieve larger $t_{overload}$ is selected. If they are also identical, then a processor will be selected randomly. For the task, which can not be allocated by the chief scheduling algorithm, the rescheduling algorithm will try to move the previously scheduled tasks and find available time slots. If a task still can not find its available allocation, it will be rejected. Before scheduling a task, the validity checking is performed to guarantee the validity of allocation.

4.3.1 Chief scheduling algorithm

1. On a new task arrival
   1) Within $[r_i, d_i]$, on each processor, if the AP of the primary pr_i and the AP of the backup bk_i for the new task T_i exist and pass the validity checking,
      i. schedule pr_i and bk_i to the corresponding locations.
      ii. set $Win_s(pr_i)$ and $Win_s(bk_i)$, and if the new task is overloaded on a previously scheduled task set τ, update $Win_s(\tau)$.
   2) If any one AP does not exist,
      i. call rescheduling algorithm
      ii. if both the AP of pr_i and the AP of bk_i exist after the rescheduling and pass the validity checking,
         a. pr_i and bk_i are scheduled,
         b. set $Win_s(pr_i)$ and $Win_s(bk_i)$, and if the new task is overloaded on a previously scheduled task set τ, update $Win_s(\tau)$.
      iii. if the AP of pr_i or the AP of bk_i does not exist after rescheduling, or any one of them can not pass the validity checking,

a. reject the task T_i.

2. Set and update shift window

1) If the task T_i is not overloaded on any other tasks,

i. if the task is a primary pr_i,

a. $Win_s(pr_i) = \langle st(pr_i) - r_i,\ st(bk_i) - ft(pr_i) \rangle$.

ii. if the task is a backup bk_i,

a. $Win_s(bk_i) = \langle st(bk_i) - ft(pr_i),\ d_i - ft(bk_i) \rangle$.

iii. the size of overloading task set is 1 for each task and the shift window of each task set is the same as that of this task.

2) If the task is overloaded on an overloading task set τ, and $Win_s(\tau)$ is <*back, for*>

i. if the task is a primary pr_i,

$$Win_s(pr_i).back = min\left[st(pr_i) - r_i,\ Win_s(\tau).back \right],$$

$$Win_s(\tau).back = Win_s(pr_i).back,$$

$$Win_s(pr_i).for = min\left[st(bk_i) - ft(pr_i),\ Win_s(\tau).for \right],$$

$$Win_s(\tau).for = Win_s(pr_i).for.$$

ii. if the task is a backup bk_i;

$$Win_s(bk_i).back = min\left[st(bk_i) - ft(pr_i),\ Win_s(\tau).back \right],$$

$$Win_s(\tau).back = Win_s(bk_i).back,$$

$$Win_s(bk_i).for = min\left[d_i - ft(bk_i),\ Win_s(\tau).for \right],$$

$$Win_s(\tau).for = Win_s(bk_i).for.$$

iii. update the shift window of the other task in τ.

4.3.2 Rescheduling algorithm

In the rescheduling algorithm, an overloading task set is moved as a whole. After the rescheduling, the relationship of the overloaded tasks can not be changed.

1. Re-scheduling

1) T_i is the task to be scheduled; τ_j is a previously scheduled overloading task set between r_i and d_i. τ_j can include one or more tasks. The $Win_s(\tau_j)$ is <$back_j$, for_j>. The number of all τ_j is s, that is $1 \le j \le s$.

2) If $back_j + for_{j+1} < c_i$, then enlarge properly the interval between two neighboring task sets τ_j and τ_{j+1} by moving them backward and forward respectively to find a time slot for T_i. (If $st(\tau_1)$ is less than r_i, or $ft(\tau_s)$ is larger than d_i , then move τ_1, τ_s both forward or backward to find an available time slot.)

3) If a time slot is available, calculate $AP(pr_i, p_j)$ of this processor.

4) repeat 1) to 3) on each processor for all τ_j and τ_{j+1}.

5) If AP exists, then return AP. Otherwise, reject T_i.

2. Moving

1) If try to move the overloading task set τ_j forward and the forward time is t.

i. if $Win_s(\tau_j).for + t \ge st(\tau_{j+1})$, iteratively move τ_{j+1} forward with the forward time

$$t = Win_s(\tau_j).for + t - st(\tau_{j+1}),$$

ii. if $Win_s(\tau_j).for + t < st(\tau_{j+1})$, the moving forward is successful. Otherwise the moving is failed.

2) If try to move the overloading task set τ_j backward and the backward time is t.

i. if $Win_s(\tau_j).back - t \le ft(\tau_{j-1})$, iteratively move τ_{j-1} backward with the backward time

$$t = ft(\tau_{j-1}) - Win_s(\tau_j).back + t,$$

ii. if $Win_s(\tau_j).back - t > ft(\tau_{j-1})$, the moving is successful. Otherwise, the moving is failed.

3) If the moving is successful eventually, change the previous schedule. Otherwise, keep the previous schedule and return.

4.4. Examples

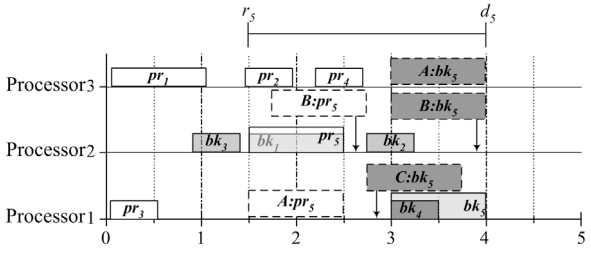


Figure 5: An example of the scheduling with AP.

Example 1.

T_1, T_2, T_3, and T_4, are previously scheduled tasks. Now, T_5 is going to be scheduled. After T_5 is scheduled, the schedule is shown in Figure 5. c_5 is 1. r_5 is 1.5. d_5 is 4. All the other values follow the scale in the figure. The primary pr_5 can be scheduled on Processor 1 and Processor 2 and has three candidate locations, on Processor 1 within the time 1.5 to 2.5 ($A:pr_5$), on Processor 2 within the time 1.75

to 2.75 (*B: pr_5*), and on Processor 2 within the time 1.5 to 2.5(*pr_5*). *pr_5* can not be scheduled on Processor 2 within the time 2 to 3 because of the overloading constraint 3. The three results of Eq.1 are

$$AP\left(pr_5, p_1, 2.5\right) = \frac{4-2.5}{4-1.5} \times \frac{1}{3} = 0.2$$
$$AP\left(pr_5, p_2, 2.75\right) = \frac{4-2.75}{4-1.5} \times \frac{2}{3} \times \frac{0.75}{1} = 0.25 .$$
$$AP\left(pr_5, p_2, 2.5\right) = \frac{4-2.5}{4-1.5} \times \frac{2}{3} \times \frac{1}{1} = 0.4$$

Thus *AP* of *pr_5* is 0.4, and *pr_5* is finally scheduled to Processor 2 with its finish time 2.5.

Example 2.

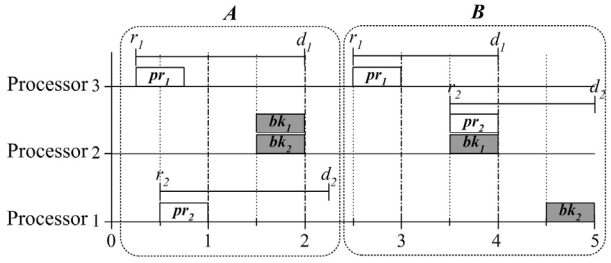


Figure 6: Examples of task set and shift window.

The task set and the change of shift window when a new task T_2 is allocated are shown in Figure 6. In the case of *A*, before T_2 is scheduled, the shift window of *pr_1* is <0, 0.75> and the shift window of *bk_1* is <0.75, 0>. After T_2 is scheduled, *bk_1* and *bk_2* form an overloading task set and have the same shift window. The shift window of *bk_1* and *bk_2* is also same with the shift window of the task set, which is <0.5, 0>. In the case of B, *pr_2* and *bk_1* form the task set. Before overloading, the shift window of *bk_1* is <0.5, 0>. After overloading, the shift window of *bk_1* is <0, 0>. The more overloading, the smaller the shift window is.

Example 3.

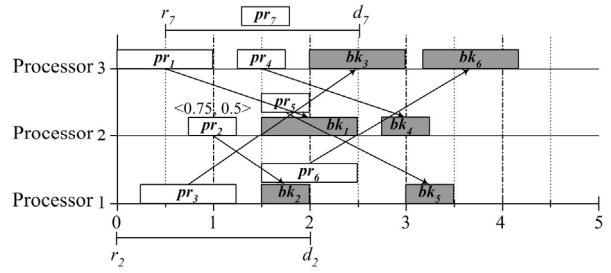


Figure 7: An example of the rescheduling.

An example of the rescheduling is shown in Figure 7. Because of the task deallocation, there exists an empty time slot before *pr_2*. This is reasonable because the deallocated task in front of *pr_2* might be a backup which is the tail of a PB chain with the maximum length or a task whose actual computation time is much less than its worst case computation time, so in the previous schedule *pr_2* can not be overloaded on this empty time slot. The newly arriving task is T_7. The *pr_7* can not be scheduled without the rescheduling. The ready time, deadline, and the shift window are displayed in the figure. The rescheduling is the only choice for *pr_7*. Therefore, *pr_2* is moved to the time slot from 0 to 0.5 and *pr_7* is scheduled to the time slot from 0.5 to 1. *bk_7* has many choices for its allocation.

5. Analysis of the algorithm

In the proposed scheduling algorithm, the main work is to search the best valid *AP*. Hence, the time complexity of our scheduling algorithm is the complexity of searching *AP* for a new *pr_i* and its *bk_i*. The worst case of searching *AP* for a new task is

1. to search on all processors and fail to find a valid *AP* in the chief scheduling algorithm, then to move the previously scheduled tasks on each processor and find a valid *AP* for *pr_i*.
2. to search on all processors and fail to find a valid *AP* in the chief scheduling algorithm, then to move the previously scheduled tasks on each processor and find no valid *AP* or only one *AP* for *bk_i*.

The search operation in the chief scheduling algorithm always takes less time than that in rescheduling algorithm, since in the worst case the search operation in the chief scheduling algorithm only need to check if there are previously scheduled task sets which can be overloaded and the intervals between task sets are large enough to contain the new task.

Let ***m*** denote the number of processors. Let ***N*** denote the average number of previously scheduled task sets on a processor. The worst case computation time follows uniform distribution within [***Min_c***, ***Max_c***]. Let ***l*** denote task laxity. For task *i*, we have

$$d_i - r_i = l_i \cdot c_i .$$

In this paper, *l* follows uniform distribution within [l_{min}, l_{max}]. Because tasks are always scheduled to overload with each other tightly, we assume the time slot occupied by a task set is approximately same with a time slot occupied by a task. Thus, the maximum *N* can be represented as

$$N_{\max} = \frac{l_{\min} \cdot Min\_c + l_{\max} \cdot Max\_c}{Min\_c + Max\_c} .$$

We assume in the worst case the *N* previously scheduled task sets distribute within [r_i, d_i] of the new task T_i as in Fig. 8.

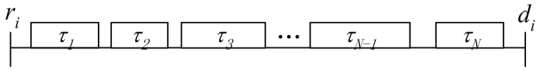


Figure 8. Previously scheduled tasks on a processor.

When the rescheduling algorithm is invoked, it means the search operation in the chief scheduling algorithm is failed. In the worst case, the primary will search all *m*

processors and the backup will search *m*-1 processors because of the constraint 2. A version of a new task will be inserted into the intervals between τ_i in Figure 8. For a new primary the number of intervals is *N*, since the last interval after τ_n can not be used by a primary (if a primary is inserted into this interval its backup can not be scheduled for constraint 3). It is similar for a new backup that only the last *N* intervals can be used except the first one. Only if the rescheduling algorithm finds a valid *AP* for the new primary and a valid *AP* for its backup successfully, the previous schedules, only on one processor, will be changed. For an available time interval, the moving operation only need to check if this time interval can be enlarged to contain the new task according to the shift windows of those previous task sets. Since the shift windows are known in advance, we assume this operation only take one unite time.

Finally, we represent the time cost for rescheduling a new primary on *m* processor as $N \cdot m$, and the time cost for rescheduling a new backup on *m*-1 processor as $N \cdot (m-1)$. The time complexity of proposed task scheduling algorithm is

$$O\left[N^2 \cdot m \cdot (m-1)\right] \ .$$

6. Simulation studies

To evaluate our task scheduling presented above, we have performed a series of simulations. We use the performance metric, ***Task Rejection Ratio*** [4, 12], to evaluate the experimental results. *Rejection Ratio (RR)* is defined to be the ratio of the number of tasks rejected to the total number of tasks that arrive at the system.

$$RR = \frac{the\ number\ of\ tasks\ rejected}{the\ number\ of\ tasks\ arrived} \qquad (10)$$

The parameters used in our simulations are summarized in Table 1.

Table 1: Parameters of simulations

Parameter	Description	Value
n	number of tasks	10000
m	number of processors	2,3,…,10
l	task laxity	3,4,…,10
Max_c	maximum computation time of tasks	100s
Min_c	minimum computation time of tasks	20s
ac_ratio	ratio of the actual to the worst case computation time	0.5~1
load	task load	0.5,0.6,…,1

Some parameters are generated as follows:

- c_i is a random number following uniform distribution between *Max_c* and *Min_c*.
- ac_i is the multiple of c_i and *ac_ratio* which follows uniform distribution.
- The deadline of a task T_i is uniformly chosen between $r_i + 2 \cdot c_i$ and $r_i + l \cdot c$. $r_i = a_i + \delta$. δ is a random number between 1s and 10s.
- The time interval of faults follows exponential distribution with the mean (*MTBF*). The minimum value is $2l \cdot Max\_c$.
- The interval between task arrivals follows exponential distribution.
- The task load is defined as the expected number of task arrivals per mean service time and its value is approximately equal to the ratio of the mean computation time of tasks to the mean time interval of task arrivals.

According to the introduction of Section 2, we compare our task scheduling algorithm with the algorithms in [4] and [8], which are named **SG** and **RA**. Because PB-overloading and BB-overloading exist in two algorithms in [8], RA includes **RAPB** and **RABB**. In order to make a fair comparison, all algorithms will share the parameter setting with our algorithm and the parameter *weight* in [4] is set to be 0. Our task scheduling approach with the rescheduling algorithm is denoted as **OR**. The rescheduling algorithm can be reduced in our task scheduling approach, and this reduced algorithm is denoted as **ONR**.

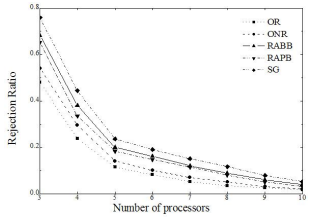


Figure 9: The relationship between *RR* and *m*. (*l* = 5, *ac_ratio* = 0.5, *load* = 0.5)

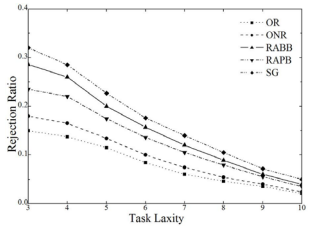


Figure 10: The relationship between *RR* and *l*. (*m* = 5, *ac_ratio* = 0.5, *load* = 0.5)

The effect of varying the number of processors on *RR* is shown in Figure 9. As the increase of the number of processors, *RR* decreases and tends to be close to 0. This is a commonness of most scheduling algorithms. The larger task laxity leads to the more flexible allocation of tasks. Hence, the larger task laxity, the smaller *RR* is. This effect is shown in Figure 10. In Figure 11, *RR* increases as task

load does. OR and ONR have the lower *RR* due to the higher processor utilization. The rescheduling can help to squeeze more empty time slots from the previous schedule, so OR is better than ONR.

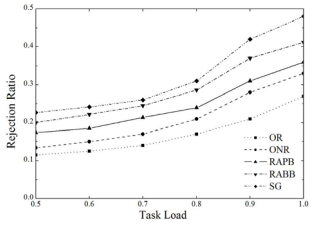


Figure 11: The relationship between *RR* and load. (*m* = 5, *ac_ratio* = 0.5, *l* = 0.5)

The maximum time length of PB chain is affected by *MTBF*. Moreover, when a fault happens, the scheduler will not schedule tasks to the failed processor. Thus, *MTBF* has the effect on *RR*. Under different task load, the effect is different. The effect of *MTBF* and *load* is shown in Figure 12. Whether OR can find and squeeze more empty time slot or not is affected by *ac_ratio*. The effect is shown in Figure 13.

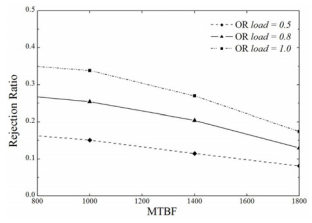


Figure 12: Effect of MTBF and load. (*m* = 5, *ac_ratio* = 0.5, *l* = 5)

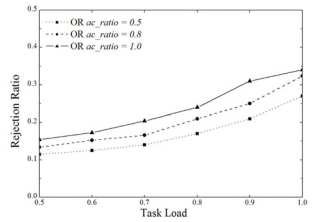


Figure 13: *RR* and different *ac_ratio*. (*m* = 5, *l* = 5)

7 Conclusion

In this paper, we have proposed a fault tolerant dynamic real-time task scheduling approach, which is based on the Primary-backup fault tolerant technique. The overloading technique in this paper is the extension of the existing ones. The scheduler uses the allocation parameter to search the proper time slots for a new task, and uses the rescheduling algorithm to change the previous task schedule for possible empty time slots by moving previously scheduled tasks on one processor. The simulations have shown that our approach is better than the others. The theoretical analysis of this scheduling approach is shown in [12].

References

[1]K. Ramamritham, J.A. Stankovic, "Scheduling algorithms and operating system support for real-time systems," *Proc. IEEE* 82 (1) pp. 55–67, Jan. 1994.

[2]G. Manimaran, C. Siva Ram Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. Parallel Distributed Systems* 9 (3) pp. 312–319, Mar. 1998.

[3]Ching-Chih Han, Kang G. Shin, and Jian Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Trans. Parallel Distributed Systems* 52(3) pp. 362–372, Mar. 2003.

[4]S. Ghosh, R. Melhem, D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. Parallel Distributed Systems* 8(3) pp. 272–284, Mar. 1997.

[5]H. Zou, F. Jahanian, "Real-time primary-backup (RTPB) replication with temporal consistency guarantees," *Proc. IEEE Intl. Conf. Distributed Computing Systems*, 1998.

[6]R. Al-Omari, A.K. Somani, G. Manimaran, "An adaptive scheme for fault-tolerant scheduling of soft real-time tasks in multiprocessor systems," *J. Parallel and Distributed Computing*, vol. 65, pp. 595-608, 2005.

[7]G. Manimaran, C. Siva Ram Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. Parallel Distributed Systems* 9 (11), pp. 1137–1152, Nov. 1998.

[8]R. Al-Omari, A.K. Somani, G. Manimaran, "Efficient overloading techniques for primary-backup scheduling in real-time system," *J. Parallel and Distributed Computing*, vol. 64, pp. 629-648, 2004.

[9]J.A. Stankovic, K. Ramamritham, "The spring kernel: a new paradigm for real-time operating systems," *ACM SIGOPS, Oper. Systems Rev.* 23 (2), pp.77–83, Jan. 1995.

[10]Y. Oh and S. H. Son, "Scheduling real-time tasks for dependability," *J. Operation Reserch Society*, 48(6) pp. 629-639, Jun. 1997.

[11]C. Shen, K. Ramamritham, and J.A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Trans. Parallel and Distributed Systems*, 4(4), pp. 382-397, Apr. 1993.

[12]W. Sun, Y. Zhang, C. Yu, X. Defago and Y. Inoguchi, "Hybrid Overloading and Stochastic Analysis for Redundant Scheduling in Real-time Multiprocessor Systems," *26th IEEE Int'l Symp. Reliable Distributed Systems*, to appear.